RECOD: A Retiming Heuristic To Optimize Resource And Memory Utilization in HW/SW Codesigns *

Karam S. Chatha
kchatha@ececs.uc.edu

Ranga Vemuri
ranga.vemuri@uc.edu

Department of ECECS, University of Cincinnati, Cincinnati, Ohio 45221-0030.

Abstract

Hardware/Software designs of embedded systems are characterized by stringent performance constraints. Pipelined implementation of a design is an effective way for maximizing the performance of a design. In this paper we present a novel retiming heuristic to obtain pipelined schedules for hardware-software codesigns. The heuristic aims at maximizing the throughput of a loop oriented resource constrained codesign while minimizing its shared memory usage. The effectiveness of the proposed technique is demonstrated by experimentation.

1 Introduction

Hardware/Software codesign process transforms a system specification into interacting hardware (HW) and software (SW) components (or tasks) which exhibit the desired behavior and satisfy the given performance constraints. A typical codesign architecture consists of a single general purpose software processor, a single application specific integrated circuit (ASIC) and a shared memory (see Figure 1), all connected through the system bus. Communication from SW to HW or HW to SW or HW to HW takes place through the shared memory. Communication between two tasks bound to SW takes place through the software memory of SW processor. The codesign partitioner binds the nodes of the graph to the SW processor or the ASIC and calls the performance evaluator. The evaluator schedules the graph on the codesign architecture and gives the partitioner feedback in terms of throughput, area and shared memory requirement estimates. The scheduler should try to maximize the throughput of the codesign while keeping the shared memory resources at a minimum. *An effective way of maximizing the throughput of a loop oriented resource constrained design is to generate a pipelined implementation. The drawback of a pipelined implementation is that it results in an increase in shared memory requirements since a number of iterations are overlapped in the steady state of the pipeline.*

Consider the task graph example shown in Figure 2. The binding and run times of the tasks are shown beside each task. Each of the data dependencies represent ten data items. Let us assume that the graph is executed a number of times in a loop. A simple non-pipelined implementation of the design is shown in the figure. The numbers in the boxes, (T,I) represent the instance of a task "T" for a particular iteration "I" of the loop. The time taken by one iteration of the loop is 325 t-units. We assume that the memory required by a task during execution is the sum of the memory occupied by its read set and write set. The memory requirement of the implementation is the maximum memory used by one iteration of the loop (shown by the dotted line in the figure)

*This work was partially supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316 and ARPA HPCC program monitored by the FBI under contract number J-FBI-93-116

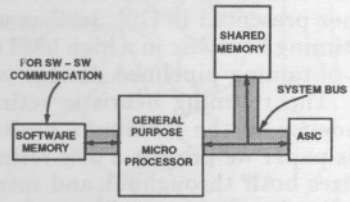

Figure 1: **Codesign Architecture**

which is 40 data items. Now consider a pipelined implementation of the same task graph. Once fully loaded the steady state completes one iteration of the loop every 200 t-units. A definite improvement over the previous design. The drawback is that the memory requirement has increased to 60 data items (shown by the dotted line).

We implement pipelined designs by using retiming transformation [6]. Given a task graph to be pipelined it can generally be retimed in more than one way. We need to select a retiming that gives us the optimal throughput with least increase in memory requirements. In this paper we present a REtiming heuristic for optimal resource utilization with least shared memory utilization of HW/SW CODesigns (RECOD).

The paper is organized as follows. In Section 2 we discuss previous work, in Section 3 we describe the system representation, Section 4 discusses the pipelined schedule, Section 5 presents RECOD, experimental results are in Section 6 and finally Section 7 concludes the paper.

2 Previous Work

The term "Retiming" was introduced by Leiserson and Saxe [6] when they used it to solve the problem of optimizing the throughput of synchronous circuitry. Since then retiming transformation has been used extensively in logic synthesis [7], high level synthesis [10], HW-SW codesign [11] and DSP applications [4]. Pipelining is considered a generalization of the retiming problem in which circuit latency is allowed to increase by allowing a change in the production and consumption times of output and input signals respectively [3]. A similar problem is "Software Pipelining" introduced by M. Lam [5]. A survey and comparison of software pipelining techniques

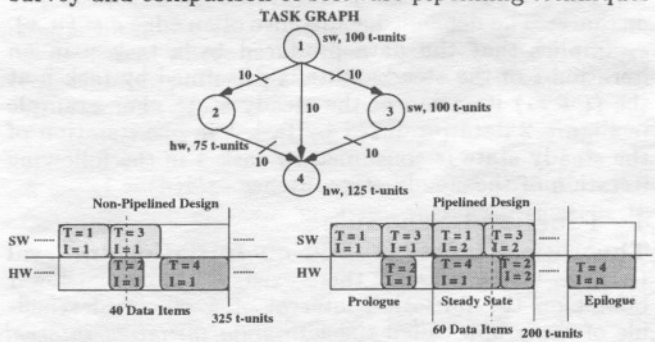

Figure 2: **Non-Pipelined and Pipelined Designs**

is given in [9]. [2] establishes a link between circuit retiming and software pipelining.

Bakshi and Gajski in [1] describe a system level design space exploration approach for pipelined implementation of HW-SW codesigns. They have used a version of a well known list scheduling algorithm to obtain pipelined schedules. The work that comes closest to the paper is that of Sánchez presented in [10]. In that work, Sánchez has used a retiming heuristic in a high level synthesis tool that aims at obtaining pipelined designs with optimum throughput. The retiming heuristic retimes the head or tail dependency of the maximum positive path in a graph. In this paper we present a new retiming heuristic which optimizes both throughput and memory requirements of pipelined codesign applications. Our heuristic does retiming in two steps. In the first step it selects a dependency to be retimed which gives the maximum freedom to the scheduler. In the second step it selects other dependencies (in addition to the first one) which on retiming result in an equivalent graph with the least increase in shared memory requirements. Experimental results show that our retiming strategy produces designs which use significantly lesser memory and operate at the optimum throughput rate.

3 System Description

The application is described as a *data dependency graph (DDG)* $G = (V, E)$, where V is the set of tasks and the edge set E, represents the data dependence between any two tasks [10]. We assume that the DDG is executed a number of times inside a loop. Associated with each task $v \in V$ are four quantities: v_{swtime}, the execution time of the task v on the general purpose SW processor, v_{hwtime} the execution of the task v in HW, v_{bind} the binding (to HW or SW) of the task which is set by the codesign partitioner and v_λ the iteration index of the task. v_{sw} can be obtained by software profiling while v_{hw} of a task is obtained by HW performance estimation [12]. The steady state of a pipeline consists of tasks belonging to different iterations of the original loop. The iteration index v_λ of a task v implies that at the i^{th} iteration of the steady state, instance of task v belonging to the $(i+v_\lambda)^{th}$ iteration of the original loop is executed. For example in Figure 2 at iteration 1 of the steady state instance of task 1 belonging to the second iteration of the original loop is executed. Hence $v_\lambda(Task1) = 1$. Since initially all tasks belong to the same iteration of the loop, v_λ for all $v \in V$ is zero. Each edge $e \in E$ has two quantities associated with it: e_{var} the number of variables transferred across a dependence and e_δ the dependence weight or dependence distance. The dependence distance of an edge $e = (u, v)$, e_δ implies that the data produced by a task u in an iteration i of the steady state is consumed by task v at the $(i + e_\delta)$ iteration of the steady state. For example in Figure 2 data produced by task 1 in one iteration of the steady state is consumed by task 4 in the following iteration of the steady state. Hence $e_\delta(1, 4) = 1$.

4 Pipelined Schedule

Theoretical Lower Bound on Initiation Interval (MII): Given a DDG there exists a theoretical lower bound on the initiation interval of a pipelined schedule of the graph called the *minimum initiation interval (MII)* [10]. For a particular pipeline implementation the *initiation interval, II*, is the time required for one iteration of the steady state. The MII is limited by two factors. Firstly the number of resources (HW or SW) limit MII. This is called the *resource constrained MII, ResMII*. The $ResMII_{HW}$ ($ResMII_{SW}$) due to the HW (SW) resource is given by the sum of the execution times of all the tasks bound to the HW (SW) resource. The execution time of a task u, u_{exec}, is the total execution time of the task. It is the sum of the task's read time, run time on the particular resource that its been bound to and write time. The read (write) time of a task is the product of the number of variables read (written) by the task and the memory read (write) time. $ResMII$ for the DDG is then the maximum of $ResMII_{HW}$ and $ResMII_{SW}$. Secondly recurrences or cycles in a task graph also limit MII. This is called the *recurrence constrained MII, RecMII*. The $RecMII_r$ for a recurrence r, is given by the ratio of the sum of the latencies of the tasks in the recurrence to the sum of the weights (e_δ) of all the dependencies in a recurrence [10]. A graph may have more than one cycle, and $RecMII$ is then the maximum of the $RecMII_r$ due to each one of them. The MII is then the maximum of $ResMII$ and $RecMII$ ($MII = \max(ResMII, RecMII)$).

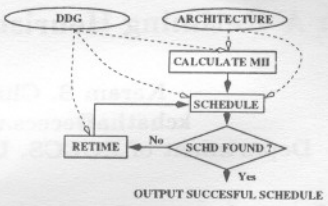

Figure 3: **Pipelined Scheduling Flow Diagram**

Pipelined Schedule: The pipelined schedule with an initiation interval II is an assignment of start times to tasks, $S(v)$, such that for all tasks v in the graph $0 \leq S(v) \leq II$ [10]. For a dependency $e = (u, v)$, the schedule time of u and v must honor the data dependence, that is $S(v) + e_\delta \times II \geq S(u) + u_{exec} \Rightarrow S(v) \geq S(u) + u_{exec} - e_\delta \times II$. We obtain a pipelined schedule by scheduling and retiming in an iterative manner as shown in Figure 3. We calculate the MII, and try scheduling the DDG for MII. However due to constraining dependencies we may not be able to schedule the DDG in MII. If we can't, we retime the DDG and try again. *The objective of retiming is to reduce the number of schedule constraining dependencies.*

Schedule constraining dependencies: A dependency $e = (u, v)$ with $e_\delta = 0$ implies that the data produced by the predecessor task u is consumed by the successor task v in the same iteration of the steady state and hence it constrains the schedule. Such a dependency is called a *intra loop dependency (ILD)*. We assume that the all the tasks belonging to one iteration of the steady state are executed before any task belonging to the next iteration. Also the HW and SW resources are themselves non-pipelined with respect to task execution. Then a dependency $e = (u, v)$ with $e_\delta > 0$ does not constrain the pipelined schedule since for all values of $S(u)$ and $S(v)$ the data dependence is satisfied. Such a dependency

is called a *loop carried dependency (LCD)*. Basically a LCD represents a data dependence between tasks belonging to different iterations of the steady state. Hence the set of schedule constraining dependencies, E^S is give by $E^S = \{e = (u, v) \in E | e_\delta = 0\}$.

A path $p = \{e_1, ..., e_n\}$ in the DDG is called a constraining path, if $\forall e \in p$, e is a schedule constraining dependency. The length of p is given by, $Length(p) = (w_{exec} + \sum_{(u,v) \in p} u_{exec})$ where w_{exec} is the execution time of the tail task of p. A critical path CP in the DDG is a constraining path p, such that for any other constraining path $p\prime \subseteq E$, $Length(p) \geq Length(p\prime)$. The length of the critical path is called the critical path time, CPT of the DDG. For a feasible pipelined schedule of the DDG with initiation interval II, $CPT \leq II$. *Hence during retiming we should try to reduce the number of schedule constraining dependencies which belong to a longer constraining path.*

Calculation of memory requirement: $LCDs$ represent data dependencies between tasks belonging to different iterations of the steady state. Hence before an iteration of the steady state can begin there is already some memory occupied by the LCD data which is given by $Mem_{LCD} = \sum_{e \in LCD} e_\delta \times e_{var}$. The memory required during one iteration of the steady state is the maximum amount of memory occupied by the data items during execution, Mem_{exec}. This memory is both due to $ILDs$ and $LCDs$. The memory requirement of a pipelined design, $MemReq$ is then given by: $MemReq = \max(Mem_{LCD}, Mem_{exec})$. As we see by the above discussion Mem_{LCD} is a lower bound on the memory requirement of a pipelined schedule. During retiming we convert a schedule constraining dependency (ILD) in to a LCD leading to an increase in shared memory requirement. *Therefore during retiming we should try to reduce the increase in* Mem_{LCD}.

Each task in the DDG is bound to a unique resource. Hence we should be able to schedule the DDG in MII time when the binding is known (and $RecMII < ResMII$). The general case where binding is unknown increases the complexity of the scheduler. However, the retiming heuristic should work equally well in the general case.

5 RECOD: Retiming Heuristic

We do retiming when we are unable to schedule a DDG in the given initiation interval, II. Retiming transformation converts an ILD into a LCD which does not constrain the scheduler. It produces an equivalent DDG with tasks belonging to different iterations, thereby pipelining the DDG. Two graphs, $DDG = G(V, E)$ and $DDG' = G(V', E')$ (obtained after retiming) are equivalent if, $\forall e = (u, v) \in E, E'$, the following equation holds, $v_\lambda - u_\lambda + e_\delta = v'_\lambda - u'_\lambda + e'_\delta$.

The drawback of retiming is that it increases the memory requirement of the schedule. We can minimize this increase by using good heuristics to select the dependency to be retimed. But this is not enough. In order to produce an equivalent DDG other dependencies might need to be retimed. The increase in shared memory requirement due to these dependencies should also be minimized. Hence RECOD does retiming in two steps. In the first step it heuristically selects a dependency to be retimed. In a DDG there might exist a number of sets of dependencies which could be retimed to obtain an equivalent DDG. In step 2 we select the set of dependencies which on retiming result in the least increase in shared memory requirement.

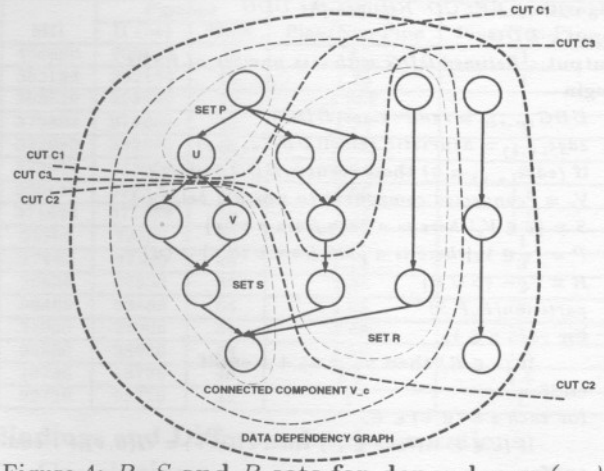

Figure 4: P, S **and** R **sets for dependency** (u,v)

RECOD Step 1: Heuristic To Select A Dependency For Retiming Transformation: The priority of a dependency to be retimed depends on its following four properties in decreasing order:

1. Dependency is an ILD:
The primary objective of RECOD is to reduce scheduling constraints in the DDG; and give the scheduler greater freedom in scheduling tasks on the resources. Since only $ILDs$ constrain the scheduler the dependency to be retimed should be an ILD.

2. Dependency between tasks bound to heterogeneous resources:
The main objective of the retiming heuristic is to reduce scheduling constraints in the graph. Increasing the distance of a dependency between tasks mapped to the same resource does not necessarily help the scheduler. Basically the two tasks have to be scheduled on the same resource and will be scheduled one after the other. On the other hand retiming a dependency between tasks mapped to different resources definitely gives more freedom to the scheduler.

3. Dependency whose predecessor task belongs to a longer constraining path.
As discussed in the previous section the constraining paths limit the II of a pipeline schedule. Retiming a dependency whose predecessor task belongs to a longer constraining path helps in obtaining a pipelined schedule with smaller II.

4. Dependency representing the least number of variables transferred.
A secondary objective of retiming transformation is to minimize the increase in memory requirement of the DDG. Increasing the distance of a dependency with more variables definitely results in a larger increase in memory requirement. Hence we select a dependency representing fewer variables being transferred.

We use property 1 to select dependencies to be retimed,

Algorithm *RECOD: Retimes the DDG*
Input : *DDG*
Output : *Retimed DDG with less number of PSDs*
Begin
$DDG_{no\_scc} = remove\_scc(DDG)$
$edge_{(u,v)} = heuristic\_select(DDG_{no\_scc})$
if $(edge_{(u,v)} = 0)$ **then return**$(DDG, failure)$
$V_c = \{connected\ component\ to\ which\ u\ belongs\}$
$S = \{v \in V_c | there\ is\ a\ path\ from\ u\ to\ v\}$
$P = \{v \in V_c | there\ is\ a\ path\ from\ v\ to\ u\} \cup \{u\}$
$R = V_c - \{S \cup P\}$
$partition(R, P, S)$
for *each* $u \in V_c$
if $(u \in P)$ **then** $u_\lambda = u_\lambda + 1$ **endif**
endfor
for *each* $e = (u, v) \in E_c$
if $(u \in P\ AND\ v \notin P)$ **then** $e_\delta(u, v) = e_\delta(u, v) + 1$ **endif**
endfor
$copy\_changes(DDG_{no\_scc}, DDG)$
return$(DDG, success)$
end

Figure 5: **RECOD: Algorithm**

and use properties 2 , 3 and 4 (in that order) to break ties.

RECOD Step 2: Partitioning To Minimize Increase In Shared Memory Requirement: In step 2 we select the set of dependencies which give us the least increase in memory requirement. Given a dependency $e = (u, v)$ (selected in step 1) to be retimed we define the following four sets with respect to u:
$V_c = \{connected\ component\ to\ which\ u\ belongs\ \}$
$P = \{w \in V_c | there\ is\ a\ path\ from\ w\ to\ u\ \} \cup \{u\}$
$S = \{w \in V_c | there\ is\ a\ path\ from\ u\ to\ w\ \}$
$R = V_c - \{P \cup S\}$
Figure 4 gives an illustration of the four sets. We can retime the dependency $e = (u, v)$ by the following three equations.
$u_\lambda = u_\lambda + 1$
$e_\delta(u, x) = e_\delta(u, x) + 1, \forall x \in V$ such that $(u, x) \in E$
$e_\delta(x, u) = e_\delta(x, u) - 1, \forall x \in V$ such that $(x, u) \in E$
Application of the three equations would result in an equivalent DDG. However the third equation decreases the distance of some dependencies. Decreasing the e_δ of a dependence is likely to change it in to a ILD. This can be avoided by increasing the u_λ of all tasks u which are in P, that is $\forall u \in P, u_\lambda = u_\lambda + 1$. Now to obtain an equivalent DDG we need to increase the e_δ of all dependencies e whose predecessor task is in the set P, but successor isn't, that is $\forall (u, v) \in E, u \in P, v \notin P, e_\delta(u, v) = e_\delta(u, v) + 1$. This is the *cutset c1* in Figure 4. Another way to retime without decreasing the e_δ of any dependence is as follows, $\forall u \in \{P \cup R\}, u_\lambda = u_\lambda + 1$ and $\forall (u, v) \in E, u \notin S, v \in S, e_\delta(u, v) = e_\delta(u, v) + 1$. This is the *cutset c2* in Figure 4. However it is possible that neither cutset $c1$ nor $c2$ result in a minimum increase in memory. We could obtain another *cutset c3* (see Figure 4) by partitioning the set R into P and S, so that the memory increase is minimized. We use a *simulated annealing* based partitioner. The cost function being minimized is defined as follows. For a cut $c_i = \{e_1, e_2, \ldots, e_n\}$, the cutsize cost is given by : $Cost = \sum_{j=1}^{n} var(e_j)$, where $var(e_j)$ is the number of variables across the dependency e_j. In the cost function the sum gives us the extra memory required by the $LCDs$ after retiming. During partitioning we ensure that if a task u is in partition P (S) then all its predecessors (successors) are also in partition P (S). After partitioning set R in to sets P and S we do retiming using the following two equations:
$\forall u \in P, u_\lambda = u_\lambda + 1$
$\forall e = (u, v) \in E, u \in P, v \notin P, e_\delta(u, v) = e_\delta(u, v) + 1$

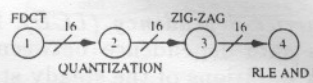

Figure 6: **DDG for JPEG like Algorithm**

	FDCT	Quant.	ZigZag	RLE & Huff.
SW time(ns)	371300	7560	1630	18480
HW time(ns)	8400	600	400	88400

Table 1: **SW and HW run times for JPEG tasks**

RECOD: Algorithm: The algorithm to do retiming transformation is shown in figure 5. A brief explanation of the functions used in the algorithm are as follows. The function *remove_scc()* replaces every strongly connected component, scc_i (or recurrence) in the DDG with a single task u_{scc_i}. It returns a new graph DDG_{no_scc}. All the dependencies that are part of a recurrence scc_i are not present in DDG_{no_scc}. All the dependencies that are "to" and "from" any task in the scc_i are now from the single task u_{scc_i}. We use DDG_{no_scc} for retiming. The distance of a dependency belonging to a recurrence in the DDG cannot be increased without decreasing the distance of any other dependency. By removing all the scc tasks and dependencies we ensure that no dependency belonging to a recurrence is retimed; although the u_λ of all the tasks belonging to a recurrence might be increased. The changes are reflected in the original DDG by the function *copy_changes()*. The function *heuristic_select()* heuristically selects a dependency to be retimed (see RECOD step 1). The function *partition()* as the name suggests partitions R between P and S (see RECOD step 2). The two *for-loops* do the retiming.

6 Experimental Results

To demonstrate the effectiveness of the retiming heuristic in HW/SW codesign, we consider the design of a JPEG [8] like compression algorithm (see Figure 6). It consists of four tasks, Forward Discrete Cosine Transform (FDCT), Quantization, Zig-Zag and RLE and Huffman encoding. All the dependencies have $e_\delta = 0$ and the number of variables transfered across each dependency is 16. The run times of the various tasks in SW and HW is shown in Table 1 [12]. Table 2 shows the estimated

No.	No. of Tasks	DDG Depth	MII	Mem. Requirements RECOD	Mem. Requirements UNRET	% Impro.
1	10	6	300	93	105	13
2	10	9	310	99	104	5
3	15	7	770	144	149	3.5
4	15	12	860	94	124	32
5	20	7	870	261	375	43.6
6	20	14	1010	205	225	9.7
7	25	4	2630	894	2767	210
8	30	5	2440	741	1732	133.7
9	50	6	5630	1984	7094	257.6
10	50	24	6410	542	565	4.2

Table 3: **Results for Random Graphs**

No.	Binding				Non-Pipeline		MII	Pipeline		Speed-up	Memory incr.
	FDCT	Quant	ZigZag	RLE & Huff.	Time (ns)	Mem		II (ns)	Mem	Pipe/Non-Pipe	Pipe/Non-Pipe
1	SW	SW	SW	SW	400890	0	400890	400890	0	1	1
2	SW	SW	SW	HW	470810	16	382154	382154	16	1.23	1
3	SW	SW	HW	SW	399660	32	398620	398620	32	1.002	1
4	SW	SW	HW	HW	469580	32	379884	379884	32	1.24	1
5	SW	HW	SW	SW	393930	32	392690	392690	48	1.003	1.5
6	SW	HW	SW	HW	463850	32	373954	373954	64	1.24	2
7	SW	HW	HW	SW	392700	32	390420	390420	48	1.06	1.5
8	SW	HW	HW	HW	462620	32	371684	371684	48	1.25	1.5
9	HW	SW	SW	SW	37990	32	29206	29206	32	1.3	1
10	HW	SW	SW	HW	107910	16	97440	97440	48	1.11	3
11	HW	SW	HW	SW	36760	32	26936	26936	64	1.36	2
12	HW	SW	HW	HW	106680	32	98480	98480	64	1.08	2
13	HW	HW	SW	SW	31030	32	21006	21006	32	1.48	1
14	HW	HW	SW	HW	100950	32	98680	98680	48	1.02	1.5
15	HW	HW	HW	SW	29800	32	18736	18736	48	1.6	1.5
16	HW	HW	HW	HW	99720	32	99720	99720	32	1	1

Table 2: **Performance Estimates for Different Bindings and Different Implementations**

run times and memory requirements for non-pipelined and pipelined implementation with different bindings of the tasks. The sixth and seventh columns have the run time and memory requirement of the non-pipeline design. Columns eight, nine and ten respectively give the MII, achieved II and memory requirement of the pipelined implementation. In the table we have exhaustively bound all the tasks to SW and HW. The results show that we were always able to schedule the DDG in MII time. We can achieve a speed-up of upto 1.6 (row 15). The maximum speed-up is low because of the non-uniform run times of the tasks. Given uniform task times the theoretical upper bound on the speed-up of a linear graph due to pipelining on two resources is 2. Depending on the throughput requirements the pipeline implementation in row 13 may be considered a better design than row 15. The pipelined design in row 13 uses the same memory as the non-pipeline implementation and gives a speed-up of 1.48.

We next demonstrate the improvement due to RECOD over an existing retiming heuristic. We compare against UNRET [10] which retimes either the head dependency of the MPP or the tail dependency of the MPP. We used random graphs and random HW/SW bindings for this experiment. The results are shown in Table 3. With both the heuristics we were able to achieve pipeline schedules in MII time (hence we do not have separate columns for II). The percentage improvement in memory requirements due to RECOD varies from 3.5 to over 257 percent. RECOD performs better than UNRET primarily because of two reasons. Firstly RECOD uses better heuristics (RECOD step 1) to select a dependency which gives maximum freedom to the scheduler. As a result it takes fewer retiming steps and hence lesser shared memory to generate an optimal schedule. Secondly RECOD uses heuristics (RECOD step 2) to select dependencies which on retiming give an equivalent DDG with lesser increase in shared memory, whereas UNRET doesn't. When the number of tasks in the graph are large and the depth of the graph is small (rows 7,8 and 9) RECOD gives significantly better results as compared to UNRET. For such graphs the algorithm can explore a large portion of the graph and hence produce better results.

7 Conclusion

We have presented a novel retiming heuristic for optimized resource and memory utilization in HW/SW codesigns. The advantage of using this heuristic is that we can obtain pipelined designs with optimal throughput, using slightly more (and in some cases same) shared memory as compared to non-pipelined design. The disadvantage is the potential increase (up to three fold) in shared memory requirements. We have incorporated this heuristic in to a codesign kernel which partitions and schedules designs for pipelined implementation.

References

[1] S. Bakshi and D.D. Gajski, "HW/SW Partitioning and Pipelining," *Proceedings of 34th DAC*, Anaheim, CA, June 1997.

[2] P. Calland, A. Darte and Y. Robert, "A new guaranteed heuristic for the software pipelining problem," *Research Report No. 2759*, INRIA, France, December 1995.

[3] B. Fluiter, E.H.L. Aarts, J.H.M. Korst, W.F.J. Verhaegh and A.V.D. Werf, "The complexity of Generalised Retiming Problem," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 11, November 1996.

[4] S. Huang and J. Rabaey, "Maximizing the Throughput of High Performance DSP Applications using Behavioral Transformations," *Proceedings of EDAC-ETC-EUROASIC '94*, pp 25-40, March 1994.

[5] M. Lam, "Software Pipelining: An effective scheduling technique for VLIW Machines", *ACM SIGPLAN*, 1988.

[6] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5-35, 1991.

[7] S. Malik, K.J. Singh, R.K. Brayton and A. Sangiovanni-Vincentelli, " Performance Optimization of Pipelined Logic Circuits Using Peripheral Retiming and Resynthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol 12, No. 5, May 1993.

[8] W.B. Pennebaker and J.L.Mitchell, "JPEG: Still Image Data Compression Standard," *Van Nostrand Reinhold*, 1993.

[9] J. Ruttenberg, G.R. Gao, A. Stoutchinin and W. Lichtenstein, "Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler," *ACM SIGPLAN NOTICES*, May 1996.

[10] F. Sánchez, "Loop Pipelining With Resource And Timing Constraints," Ph.D. Dissertation, UPC Universitat Politècnica de Catalunya, Barcelona, Spain, October 1995.

[11] M. Sheliga, N.L. Passos and E.H. Sha, "Fully Parallel Hardware/Software Codesign For Multi-Dimensional DSP Applications," *Proceedings of Codes/CASHE '96* , March 1996.

[12] J. Walrath, Karam S. Chatha, R. Vemuri, N. Narasimhan and V. Srinivasan, "Performance Modeling and Tradeoff Analysis During Rapid Prototyping," *Proceedings of the 1996 International Conferences on Application-Specific Systems, Architectures and Processors*, IEEE press, August 1996.